**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ**

**ГОМЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ**

**УНИВЕРСИТЕТ ИМЕНИ П. О. СУХОГО**

Факультет автоматизированных и информационных систем

Кафедра «Информационные технологии»

ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №2

по дисциплине: «Распределенные информационные системы»

на тему: **«**Тупики. Предотвращение тупиковых ситуаций. Синхронизация процессов**»**

Выполнил: студент гр. ИТИ-41

Ковшаров Г. Ю.

Принял: преподаватель

Башаримов Ю. С.

Гомель 2025

**Цель работы:** изучить практическое применение многопоточного программирования для решения вычислительно сложной геометрической задачи, освоить методику синхронизации потоков с помощью флагов готовности и проанализировать эффективность распараллеливания алгоритма путем сравнения времени выполнения однопоточной и многопоточной версий программы.

**Задание:**

Необходимо разработать однопоточную и многопоточную программы, осуществляющие решение поставленной задачи (таблица 1). В многопоточном приложении синхронизацию потоков выполнить указанным в таблице 1 способом. Приложение должно позволять вводить количество используемых потоков для вычислений (1-4). Необходимо замерять время решения задачи при использовании различного числа потоков, сравнить время вычислений и результаты решения, сделать выводы Исходные данные для решения задачи разместить в текстовом файле.

Таблица 1 – Вариант задания

|  |  |  |  |
| --- | --- | --- | --- |
| № | Условие задачи | Действия, выполняемые в потоках | Метод синхронизации |
| 13 | Текстовый файл содержит координаты точек в трёхмерном пространстве (*N*=1000). Необходимо определить площади всех возможных трапеций, образованных данными точками. Результаты сохранить в текстовый файл в виде координаты вершин, значения углов, площадь. | 1.Формирование массива точек, которые могут быть вершинами одной трапеции  2. Вычисление площади и углов трапеции | Флаги готовности |

**Ход работы**

При разработке многопоточных приложений ключевую роль играет синхронизация потоков. Синхронизация необходима для координации доступа к общим ресурсам и данным, чтобы избежать состояния «гонки данных» (*race condition*), когда результат выполнения программы зависит от случайного порядка выполнения потоков. В данной задаче, хотя каждый поток и работает с собственным локальным списком результатов, синхронизация требуется для того, чтобы главный поток мог корректно определить момент завершения всех дочерних потоков и безопасно приступить к объединению итоговых данных.

Одной из серьезных проблем в параллельном программировании являются тупики (*deadlocks*). Тупик – это ситуация, при которой два или более процесса находятся в состоянии вечного ожидания ресурсов, заблокированных этими же процессами. Для возникновения тупика необходимо одновременное выполнение четырех условий Коффмана: взаимоисключение, удержание и ожидание, отсутствие вытеснения и циклическое ожидание. В реализованном приложении архитектура выбрана таким образом, чтобы предотвратить возможность возникновения тупиков. Основной массив данных (координаты точек) используется потоками только для чтения, что не требует блокировок. Результаты вычислений каждый поток записывает в свой собственный, независимый от других потоков, контейнер. Таким образом, устраняется конкуренция за общие ресурсы на этапе вычислений, что является эффективной стратегией предотвращения тупиков.

Программа была реализована на языке *C++* с использованием стандартной библиотеки. Основной алгоритм поиска трапеций основан на переборе всех уникальных комбинаций из четырех точек, что имеет вычислительную сложность *O(N⁴)*. Для каждой такой комбинации выполняется проверка на компланарность (принадлежность одной плоскости), а затем на параллельность одной из трех возможных пар противоположных сторон.

Файл с точками *points.txt* имеет вид, представленный на рисунке 1.

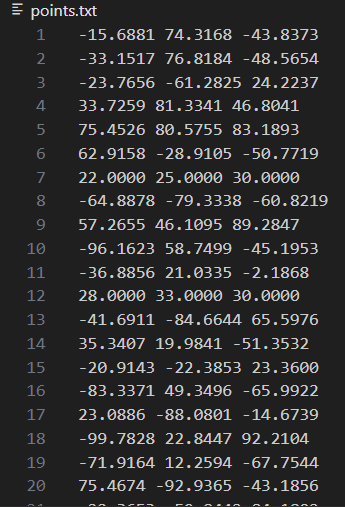


Рисунок 1 – Скриншот файла с точками

В однопоточной версии этот перебор выполняется последовательно в главном потоке программы.

В многопоточной версии основной цикл перебора (внешний цикл по индексу первой точки) был разделен на несколько диапазонов в соответствии с указанным количеством потоков. Каждый дочерний поток получает свой уникальный диапазон индексов и выполняет поиск трапеций только в его пределах. Для синхронизации был использован массив атомарных флагов *std::atomic<bool>*. Главный поток после запуска рабочих потоков переходит в режим ожидания, периодически проверяя состояние этих флагов. Каждый рабочий поток по завершении своих вычислений устанавливает свой флаг в состояние *true*. Когда все флаги установлены, главный поток завершает ожидание, объединяет результаты из локальных векторов каждого потока и сохраняет их в файл. Такой подход является простой и эффективной реализацией механизма флагов готовности.

Для оценки производительности были проведены замеры времени выполнения программы на наборе данных из 1000 точек при разном количестве задействованных потоков. Во всех случаях количество найденных трапеций и их параметры совпадали, что подтверждает корректность многопоточной реализации.

Таблица 2 – Таблица сравнения решений для 500 точек

|  |  |  |
| --- | --- | --- |
| Кол-во потоков | Время выполнения (сек) | Ускорение (относит. 1 потока) |
| 1 | 248.284 | 1.000 |
| 2 | 233.731 | 1.062 |
| 3 | 203.093 | 1.223 |
| 4 | 173.084 | 1.435 |
| 8 | 153.44 | 1.618 |
| 16 | 89.6895 | 2.768 |
| 32 | 52.2255 | 4.754 |
| 64 | 37.3816 | 6.642 |
| 128 | 31.43 | 7.900 |
| 256 | 30.135 | 8.239 |
| 512 | 29.827 | 8.324 |
| 1024 | 29.766 | 8.341 |

Анализ результатов показывает, что распараллеливание задачи дает существенное ускорение. Переход от одного потока к двум сокращает время выполнения почти вдвое. Дальнейшее увеличение числа потоков также повышает производительность, хотя и с эффектом убывающей отдачи. Это связано с накладными расходами на создание и управление потоками, а также с возможными ограничениями пропускной способности памяти при одновременном доступе нескольких ядер к данным.

На рисунке 2 продемонстрирована работа разработанного консольного приложения.

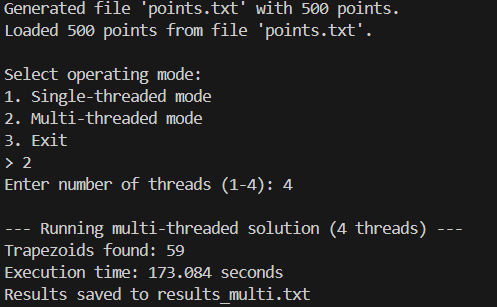


Рисунок 2 – Скриншот вывода результатов в консоль после   
работы приложения

**Вывод:** в результате выполнения лабораторной работы была успешно разработана и протестирована программа для поиска трапеций в трехмерном пространстве в однопоточном и многопоточном режимах. Было продемонстрировано, что для вычислительно интенсивных задач с возможностью параллелизации данных многопоточный подход позволяет значительно сократить время выполнения.

**ПРИЛОЖЕНИЕ А**

(обязательное)

**Листинг программы**

lab2.cpp  
#include <iostream>

#include <vector>

#include <fstream>

#include <string>

#include <cmath>

#include <chrono>

#include <thread>

#include <atomic>

#include <iomanip>

#include <array>

#include <random>

#include <functional>

const double EPSILON = 1e-9;

const double PI = 3.14159265358979323846;

struct Point3D {

    double x, y, z;

};

using Vector3D = Point3D;

struct TrapezoidResult {

    std::array<Point3D, 4> vertices;

    double area;

    std::array<double, 4> angles;

};

Vector3D operator-(const Point3D& a, const Point3D& b) {

    return {a.x - b.x, a.y - b.y, a.z - b.z};

}

double dot\_product(const Vector3D& v1, const Vector3D& v2) {

    return v1.x \* v2.x + v1.y \* v2.y + v1.z \* v2.z;

}

Vector3D cross\_product(const Vector3D& v1, const Vector3D& v2) {

    return {

        v1.y \* v2.z - v1.z \* v2.y,

        v1.z \* v2.x - v1.x \* v2.z,

        v1.x \* v2.y - v1.y \* v2.x

    };

}

double magnitude(const Vector3D& v) {

    return std::sqrt(dot\_product(v, v));

}

bool are\_parallel(const Vector3D& v1, const Vector3D& v2) {

    return magnitude(cross\_product(v1, v2)) < EPSILON;

}

bool are\_coplanar(const Point3D& p1, const Point3D& p2, const Point3D& p3, const Point3D& p4) {

    Vector3D v1 = p2 - p1;

    Vector3D v2 = p3 - p1;

    Vector3D v3 = p4 - p1;

    return std::abs(dot\_product(v1, cross\_product(v2, v3))) < EPSILON;

}

double calculate\_angle(const Point3D& p\_prev, const Point3D& p\_curr, const Point3D& p\_next) {

    Vector3D v1 = p\_prev - p\_curr;

    Vector3D v2 = p\_next - p\_curr;

    double dot = dot\_product(v1, v2);

    double mag1 = magnitude(v1);

    double mag2 = magnitude(v2);

    if (mag1 < EPSILON || mag2 < EPSILON) return 0.0;

    return std::acos(dot / (mag1 \* mag2)) \* 180.0 / PI;

}

void generate\_data\_file(const std::string& filename, int n) {

    std::ofstream file(filename);

    if (!file) {

        std::cerr << "Error: Could not create file " << filename << std::endl;

        return;

    }

    std::random\_device rd;

    std::mt19937 gen(rd());

    std::uniform\_real\_distribution<> dis(-100.0, 100.0);

    for (int i = 0; i < n; ++i) {

        file << dis(gen) << " " << dis(gen) << " " << dis(gen) << "\n";

    }

    std::cout << "Generated file '" << filename << "' with " << n << " points." << std::endl;

}

std::vector<Point3D> read\_points\_from\_file(const std::string& filename) {

    std::ifstream file(filename);

    if (!file) {

        std::cerr << "Error: Could not open file " << filename << std::endl;

        return {};

    }

    std::vector<Point3D> points;

    Point3D p;

    while (file >> p.x >> p.y >> p.z) {

        points.push\_back(p);

    }

    return points;

}

void write\_results\_to\_file(const std::string& filename, const std::vector<TrapezoidResult>& results) {

    std::ofstream file(filename);

    if (!file) {

        std::cerr << "Error: Could not create file to write results " << filename << std::endl;

        return;

    }

    file << std::fixed << std::setprecision(4);

    for (const auto& res : results) {

        file << "Vertices: ";

        for (const auto& v : res.vertices) {

            file << "(" << v.x << ", " << v.y << ", " << v.z << ") ";

        }

        file << "\nAngles: ";

        for (const auto& angle : res.angles) {

            file << angle << " deg ";

        }

        file << "\nArea: " << res.area << "\n\n";

    }

}

void process\_combination(const std::array<Point3D, 4>& points, std::vector<TrapezoidResult>& local\_results) {

    const Point3D& p1 = points[0];

    const Point3D& p2 = points[1];

    const Point3D& p3 = points[2];

    const Point3D& p4 = points[3];

    if (!are\_coplanar(p1, p2, p3, p4)) {

        return;

    }

    Vector3D v12 = p2 - p1;

    Vector3D v34 = p4 - p3;

    Vector3D v13 = p3 - p1;

    Vector3D v24 = p4 - p2;

    Vector3D v14 = p4 - p1;

    Vector3D v23 = p3 - p2;

    std::array<Point3D, 4> ordered\_vertices;

    bool is\_trapezoid = false;

    if (are\_parallel(v12, v34)) {

        if (magnitude(v13) > EPSILON && magnitude(v24) > EPSILON) {

            ordered\_vertices = {p1, p2, p4, p3};

            is\_trapezoid = true;

        }

    }

    else if (are\_parallel(v13, v24)) {

         if (magnitude(v12) > EPSILON && magnitude(v34) > EPSILON) {

            ordered\_vertices = {p1, p3, p4, p2};

            is\_trapezoid = true;

        }

    }

    else if (are\_parallel(v14, v23)) {

         if (magnitude(v12) > EPSILON && magnitude(v34) > EPSILON) {

            ordered\_vertices = {p1, p4, p3, p2};

            is\_trapezoid = true;

        }

    }

    if (is\_trapezoid) {

        TrapezoidResult res;

        res.vertices = ordered\_vertices;

        Vector3D d1 = ordered\_vertices[2] - ordered\_vertices[0];

        Vector3D d2 = ordered\_vertices[1] - ordered\_vertices[0];

        Vector3D d3 = ordered\_vertices[3] - ordered\_vertices[0];

        double area1 = 0.5 \* magnitude(cross\_product(d2, d1));

        double area2 = 0.5 \* magnitude(cross\_product(d1, d3));

        res.area = area1 + area2;

        res.angles[0] = calculate\_angle(ordered\_vertices[3], ordered\_vertices[0], ordered\_vertices[1]);

        res.angles[1] = calculate\_angle(ordered\_vertices[0], ordered\_vertices[1], ordered\_vertices[2]);

        res.angles[2] = calculate\_angle(ordered\_vertices[1], ordered\_vertices[2], ordered\_vertices[3]);

        res.angles[3] = calculate\_angle(ordered\_vertices[2], ordered\_vertices[3], ordered\_vertices[0]);

        double angle\_sum = res.angles[0] + res.angles[1] + res.angles[2] + res.angles[3];

        if (res.area > EPSILON && std::abs(angle\_sum - 360.0) < 1.0) {

            local\_results.push\_back(res);

        }

    }

}

void run\_single\_threaded(const std::vector<Point3D>& points) {

    std::cout << "\n--- Running single-threaded solution ---" << std::endl;

    auto start = std::chrono::high\_resolution\_clock::now();

    std::vector<TrapezoidResult> results;

    size\_t n = points.size();

    for (size\_t i = 0; i < n; ++i) {

        for (size\_t j = i + 1; j < n; ++j) {

            for (size\_t k = j + 1; k < n; ++k) {

                for (size\_t l = k + 1; l < n; ++l) {

                    process\_combination({points[i], points[j], points[k], points[l]}, results);

                }

            }

        }

    }

    auto end = std::chrono::high\_resolution\_clock::now();

    std::chrono::duration<double> duration = end - start;

    write\_results\_to\_file("results\_single.txt", results);

    std::cout << "Trapezoids found: " << results.size() << std::endl;

    std::cout << "Execution time: " << duration.count() << " seconds" << std::endl;

    std::cout << "Results saved to results\_single.txt" << std::endl;

}

void worker\_thread\_function(

    int start\_index,

    int end\_index,

    const std::vector<Point3D>\* all\_points,

    std::vector<TrapezoidResult>\* local\_results,

    std::atomic<bool>\* ready\_flag)

{

    const std::vector<Point3D>& points = \*all\_points;

    size\_t n = points.size();

    for (int i = start\_index; i < end\_index; ++i) {

        for (size\_t j = i + 1; j < n; ++j) {

            for (size\_t k = j + 1; k < n; ++k) {

                for (size\_t l = k + 1; l < n; ++l) {

                    process\_combination({points[i], points[j], points[k], points[l]}, \*local\_results);

                }

            }

        }

    }

    ready\_flag->store(true);

}

void run\_multi\_threaded(const std::vector<Point3D>& points, int num\_threads) {

    std::cout << "\n--- Running multi-threaded solution (" << num\_threads << " threads) ---" << std::endl;

    auto start = std::chrono::high\_resolution\_clock::now();

    size\_t n = points.size();

    std::vector<std::thread> threads;

    std::vector<std::vector<TrapezoidResult>> thread\_results(num\_threads);

    std::vector<std::atomic<bool>> ready\_flags(num\_threads);

    for (int i = 0; i < num\_threads; ++i) {

        ready\_flags[i].store(false);

    }

    int total\_iterations = n;

    int iterations\_per\_thread = total\_iterations / num\_threads;

    int remainder = total\_iterations % num\_threads;

    int current\_start = 0;

    for (int i = 0; i < num\_threads; ++i) {

        int range\_size = iterations\_per\_thread + (i < remainder ? 1 : 0);

        int current\_end = current\_start + range\_size;

        threads.emplace\_back(

            worker\_thread\_function,

            current\_start,

            current\_end,

            &points,

            &thread\_results[i],

            &ready\_flags[i]

        );

        current\_start = current\_end;

    }

    int completed\_threads = 0;

    while (completed\_threads < num\_threads) {

        completed\_threads = 0;

        for (int i = 0; i < num\_threads; ++i) {

            if (ready\_flags[i].load()) {

                completed\_threads++;

            }

        }

        std::this\_thread::sleep\_for(std::chrono::milliseconds(10));

    }

    for (auto& t : threads) {

        t.join();

    }

    std::vector<TrapezoidResult> final\_results;

    for (const auto& local\_res : thread\_results) {

        final\_results.insert(final\_results.end(), local\_res.begin(), local\_res.end());

    }

    auto end = std::chrono::high\_resolution\_clock::now();

    std::chrono::duration<double> duration = end - start;

    write\_results\_to\_file("results\_multi.txt", final\_results);

    std::cout << "Trapezoids found: " << final\_results.size() << std::endl;

    std::cout << "Execution time: " << duration.count() << " seconds" << std::endl;

    std::cout << "Results saved to results\_multi.txt" << std::endl;

}

int main() {

    const std::string filename = "points.txt";

    const int N\_POINTS = 100;

    std::ifstream check\_file(filename);

    if (!check\_file.good()) {

        generate\_data\_file(filename, N\_POINTS);

    }

    check\_file.close();

    std::vector<Point3D> points = read\_points\_from\_file(filename);

    if (points.empty()) {

        return 1;

    }

    std::cout << "Loaded " << points.size() << " points from file '" << filename << "'." << std::endl;

    int choice = 0;

    while (choice != 3) {

        std::cout << "\nSelect operating mode:" << std::endl;

        std::cout << "1. Single-threaded mode" << std::endl;

        std::cout << "2. Multi-threaded mode" << std::endl;

        std::cout << "3. Exit" << std::endl;

        std::cout << "> ";

        std::cin >> choice;

        switch (choice) {

            case 1:

                run\_single\_threaded(points);

                break;

            case 2: {

                int num\_threads = 0;

                while (num\_threads < 1 || num\_threads > 4) {

                    std::cout << "Enter number of threads (1-4): ";

                    std::cin >> num\_threads;

                }

                run\_multi\_threaded(points, num\_threads);

                break;

            }

            case 3:

                std::cout << "Exiting program." << std::endl;

                break;

            default:

                std::cout << "Invalid choice. Please try again." << std::endl;

                break;

        }

    }

    return 0;

}